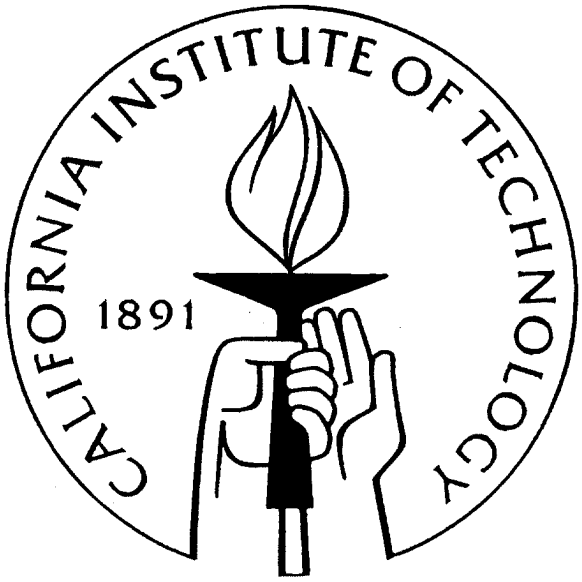

A Development Methodology for Concurrent Programs

Bryan Chow
Andy Fyfe
Daniel Maskit
Stephen Taylor
Jerrell R. Watts
Yair Zadik

Computer Science Department
California Institute of Technology

Caltech-CS-TR-94-16

A Development Methodology for Concurrent Programs [1]

Bryan Chow
Andy Fyfe
Daniel Maskit
Stephen Taylor
Jerrell R. Watts
Yair Zadik
Scalable Concurrent Programming Laboratory
California Institute of Technology

September 16, 1994

Abstract

This paper describes a development methodology for the design of concurrent programs that provides a migration path from existing sequential C and FORTRAN programs. These programs may be executed immediately, without change, using the entire physical memory of a distributed memory machine or a network of ATM-coupled shared-memory multiprocessors. Subsequent program refinements may involve data and control decomposition together with explicit message passing to improve performance. Each step in the program development may utilize new hardware mechanisms supporting shared memory, segmentation and protection. The ideas presented in this paper are currently being implemented within the Multiflow compiler which is being targetted for the M-Machine. Although the examples we present use the C programming language, the concepts will also be available in FORTRAN.

[1]The research described in this report is sponsored primarily by the Advanced Research Projects Agency, ARPA Order number 8176, and monitored by the Office of Naval Research under contract number N00014-91-J-1986.

1 Overview

The advent of low-cost shared-memory multiprocessors, coupled together using ATM technology, demands that multicomputer technology be revisited. We seek a common programming framework that can be used for both high-performance multicomputers such as the Cray T3E and MIT M-Machine [3, 4], as well as networks of shared-memory multiprocessors such as the SGI Power Challenge. In particular, we require that the framework be able to utilize new hardware mechanisms for supporting shared memory, segmentation and protection.

Our experience in working with a number of scientists on large-scale concurrent applications in Computational Fluid Dynamics, Particle Physics, Materials Science and Chemistry. In all these areas we have noticed a consistent pattern of reluctance to learn new programming concepts and discard working code. Although reprogramming is expected to express new concurrent algorithms and improve efficiency, recoding of basic sequential physics routines is viewed as error-prone and undesirable. The experience of migrating a code from a sequential implementation to a shared-memory implementation tends to enforce scientists' bias against rewriting their code to accommodate new architectural ideas.

A welcome alternative has been adopted within a networked machine in the SGI Power Challenge: sequential programs will execute immediately using the entire physical memory of the machine, which appears to the programmer as a large virtual address space. This approach has not generally been practical on distributed memory machines due to the high cost of communication and synchronization. On the SGI machine, both control and data may be subsequently decomposed in order to obtain increased concurrency and utilize multiple processors. We have experimented with coupling these machines using HIPPI connections. Unfortunately the shared memory view is only available within a single cabinet.

Our experience in programming such machines for an NSF Grand Challenge problem has been somewhat discouraging. Although it is possible to execute a sequential program immediately, it typically performs poorly. To obtain good performance, it is necessary to decompose the program data structures to obtain locality of reference. For the most part, the program that eventually results is similar to what one would expect to obtain when developing a message-passing program. The crucial difference is that scientists are able to utilize the entire memory of the machine immediately and thus obtain useful results quickly; thus they perceive less risk in this approach.

This paper describes a scalable programming methodology for parallel and distributed machines that reduces the initial risk associated with parallel programming. The methodology provides simple tools by which a programmer may incrementally design and implement a new parallel program beginning with an existing sequential program or algorithm. We do not propose, or intend to pursue, automated, compiler generated, parallel execution.

The intent of this programming methodology is to eliminate the need for programmers

to choose between parallel programming paradigms. Shared-memory programming will be supported, as will message-passing, and the programmer will be allowed to decide which paradigm is most appropriate for each section of their code. The transition between paradigms will be smoothest if programs are written so as to isolate the exchange of data between processes.

We distinguish four distinct stages of program development as depicted in Figure 1. Initially, a sequential C or Fortran program is compiled and may utilize the entire physical memory of the machine; sequential execution uses only a single computer in the network. Subsequently, control of the program is decomposed by executing the program on all nodes [5, 6]. Multiple processes are mapped to a single computer so as to overlap communication and computation. Standard message-passing constructs are added to refine and improve the program efficiency. Finally, the program can be transformed completely into a pure message-passing code [7, 8, 9].

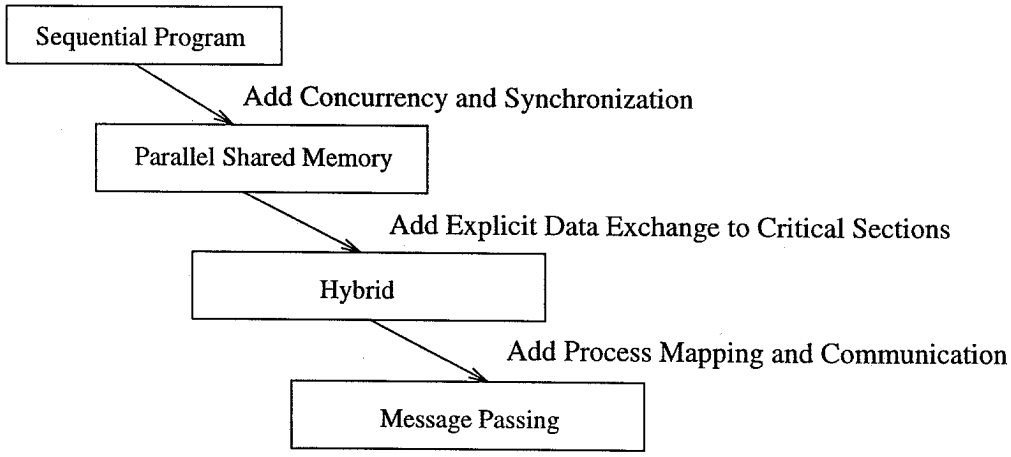

Figure 1: Stages of Program Development

This methodology can be supported on existing parallel machines such as the Cray T3D and Intel Paragon, as well as ATM coupled networks of shared memory multiprocessors. In addition, this methodology will be portable to future parallel machines. In particular, we give details of the implementation of this methodology on the MIT M-Machine, and explain how both the philosophy of the programming system and the available hardware mechanisms influence design decisions. We provide an overview of the M-Machine hardware, and describe how each stage of the methodology can be supported on this platform. This discussion emphasizes support of hardware including *tagged pointers*, *hardware synchronization*, *global virtual address space*, *V-threads*, and *H-threads*. Although collectively these concepts are unique to the M-machine, variants of the concepts can be found in other architectures including the Cray T3D. Finally, this programming model can be viewed as the target for preprocessors providing high level abstractions through the use of source-to-source transformations.

2 Abstract View of the M-Machine

This section provides an abstract description of the M-Machine hardware, with descriptions of the mechanisms that are important to support our programming methodology. The M-Machine provides four mechanisms not present in the J-Machine: a *global virtual address space* to provide a global view of data and an address space substantially larger than local physical memory, *tagged pointers* to provide memory protection and segmentation, *V-Threads* to support multi-threaded operation, and *H-Threads* to support instruction-level concurrency. Figure 2 provides an abstract view of the hardware that highlights these features.

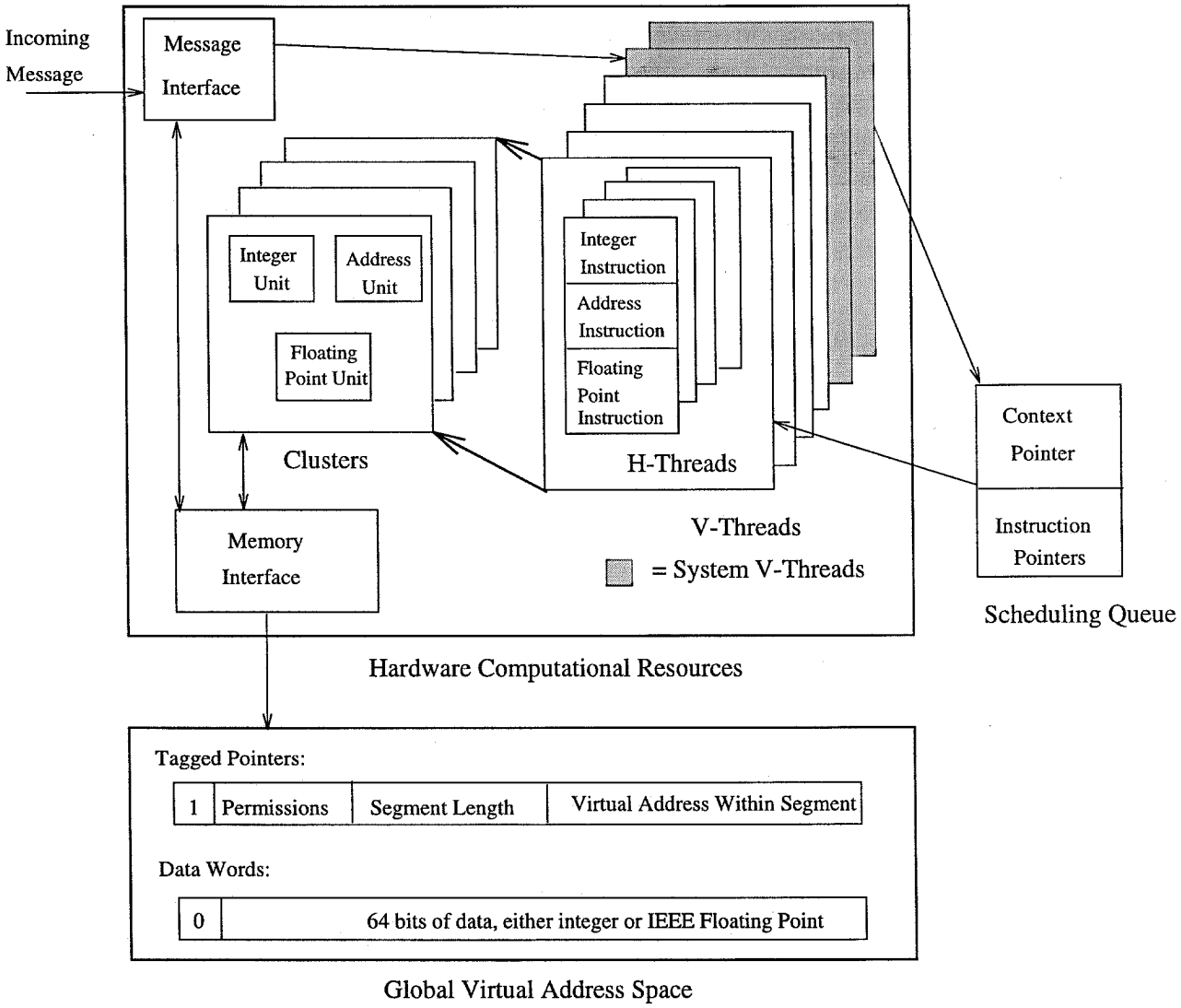

Figure 2: Abstract View of the M-Machine

Global virtual address space. This mechanism is supported through the memory interface, which is a combination of hardware and system software. This interface has

the capability of determining where a referenced virtual address physically resides. If the address is physically local, the requested operation can complete. Otherwise, the interface will generate a message to request the access from the node where it is physically resident. This interface is also in charge of managing the data cache at each node. This mechanism provides a programmable mapping of virtual to physical addresses.

Tagged pointers. These provide two different types of functionality for memory management. They allow the memory to be divided up into sized segments, and provide protection against overrunning segment boundaries. The tags also allow each pointer to have permissions associated with it. These permissions provide protection for access to and execution of system code. They also protect the memory associated with one program from that of another program. In addition, there is one bit associated with each word of memory and registers which designates whether a word is data (0) or a pointer (1).

V-Threads. Explicit hardware support for multi-threading is supported using this mechanism. A *V-Thread* is a hardware resource which consists of both data and state registers for each of the node's four clusters. There are four user *V-Thread slots* per node, as well as two system slots. The hardware manages interleaving execution of instruction streams from all slots. The system slots are used to execute kernel code to handle events and exceptions. This includes both message reception and software management of the memory interface.

When there are more threads in a node than can fit into the available slots, the runtime system will provide a scheduling queue for inactive threads. Each thread consists of a *context pointer*, which points to storage for the thread data, including the storage for saving the thread state when it is removed from a thread slot, and up to four *instruction pointers*, each of which points to the code being executed by the thread on a given cluster.

H-Threads. The superscalar architecture of the Multi-ALU Processor (MAP) [4] allows up to 12 instructions to be issued each cycle. Within each *V-Thread slot*, there are four independent *H-Thread slots* each of which controls execution at one cluster. Each of these slots has its own instruction pointer and register file. There are facilities for allowing inter-cluster communication to facilitate coordination between *H-Threads*. The compiler will use the *H-Thread slots* to exploit instruction-level parallelism.

In addition to these features, the M-Machine provides support for *hardware synchronization* and *message-driven processes* similar to that provided in the J-Machine [3].

3 Support for Sequential Programming

Sequential programming serves to utilize the entire memory of the machine.

Execution of a sequential program is achieved by providing an environment which appears, from the programmer's perspective, to be equivalent to a UNIX-style workstation

on the M-Machine. A partition of the machine, along with a portion of virtual memory, will be allocated for the job. The virtual memory will be mapped to some portion of the physical memory present on the allocated nodes, and the executable image will be downloaded into this area of memory. The program will then execute within a single thread of control on one processor. Each node within the partition could be in use by up to four different sequential jobs. Each job will be allocated 1/4, 1/2, 3/4 or all of the available memory at a given node. The selection of the partition will be guided by the total amount of memory required for the job.

3.1 Global Virtual Address Space

The global virtual address space is used to provide the appearance of the program having a single contiguous address space, while distributing the program image across some subset of the nodes in the partition for the job. After the partition is allocated, a node in the partition is designated as node 0; this node executes the sequential code. The remaining nodes are numbered consecutively from zero. If possible, all of the code and stack memory is mapped into node 0, together with some portion of the global variables. The space to be used for code and stack at node 0 will be controlled by using compiler/loader switches. Any program state not mapped to node 0 is distributed across the remainder of the nodes in the machine so as to maximize bandwidth for remote memory references as shown in Figure 3. Memory for the stack and heap is not physically mapped until needed.

3.2 Tagged Pointers

Tagged pointers will be used to provide two different types of protection. First, they will be used to prevent users from directly running or accessing system code. Secondly, they will be used to prevent jobs from interfering with one another. This will be accomplished by establishing the memory usable by the job as one segment, and ensuring that all pointers given to the job by the system are within this segment.

3.3 V-Threads

The only use of user-level V-Threads for sequential programs will be one V-thread at the node actually executing the program. There will also be some usage of system-level threads to handle the communication necessary to transport pieces of the shared memory from one node to another.

3.4 H-Threads

The compiler will do the best job it can of using all four H-Threads within the V-Thread that is executing the program. A significant amount of this usage will be attained through

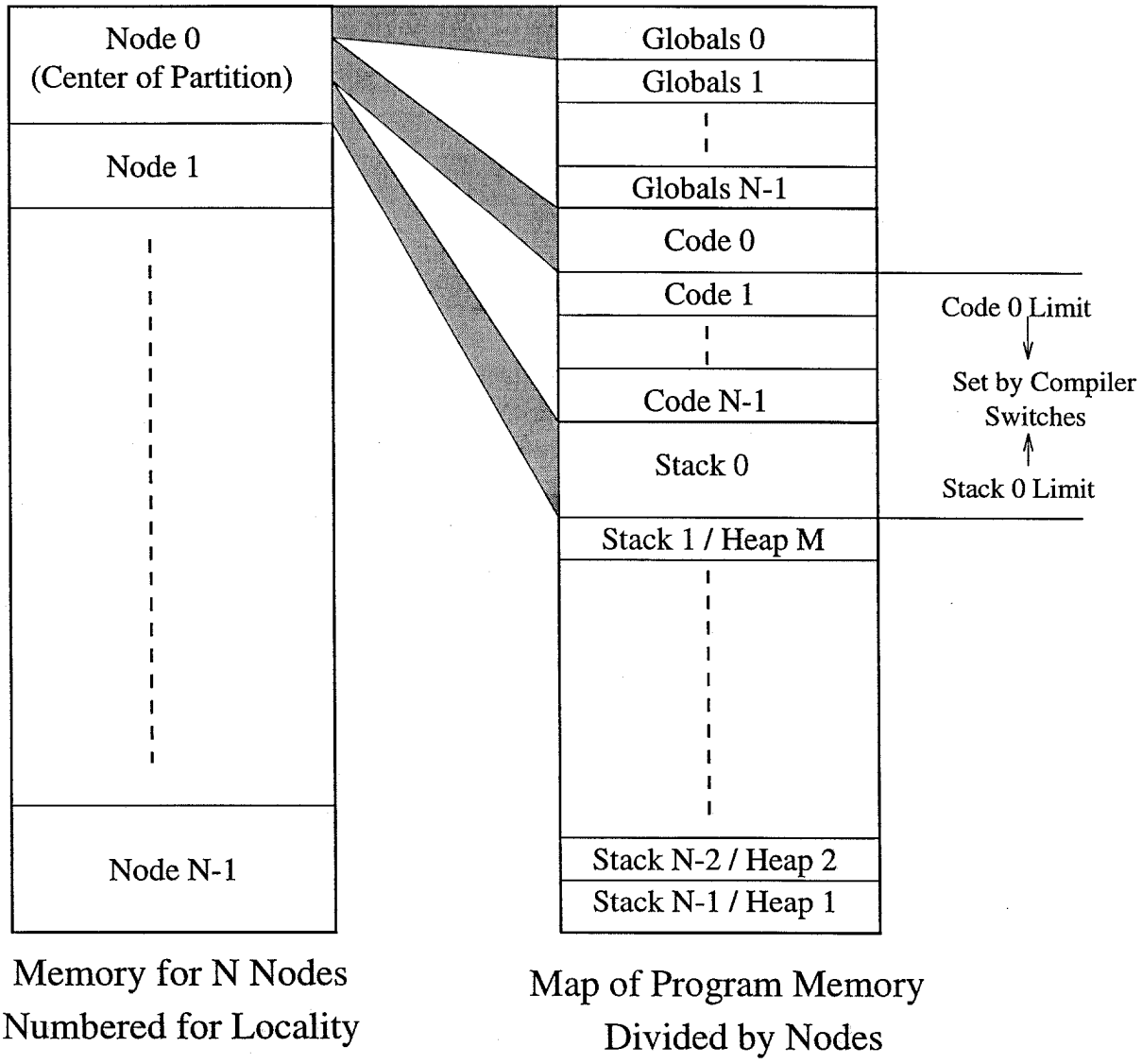

Figure 3: Mapping a Sequential Program

using loop unrolling

3.5 Example

The code shown in Program 1 is a sequential program which solves Laplace's equation $\nabla^2\theta = 0$ using constant boundary conditions. In a sequential program, the memory is distributed across the machine, but the locus of control is in one process which starts execution at **main()**.

4 Support for Shared Memory

Shared memory programming provides implicit concurrency and communication, and explicit synchronization.

The programmers first concern for parallel execution occurs with the introduction of the use of shared memory through control decomposition. This is supported by providing multiple processes, and synchronization primitives. We assume there is no dependency analysis and that the user is responsible for data consistency. To enforce consistency we utilize barrier synchronization. In the future, the concepts presented here may be used as the target for more sophisticated program analysis.

4.1 Global Virtual Address Space

As with the sequential model, the global virtual address space is used to provide the appearance of the program having a single contiguous address space, while distributing the program image across some subset of the nodes in the partition for the job. In addition, each process has its own private copies of non-shared global variables.

4.2 Tagged Pointers

In addition to the protection of system code, and of jobs from one another, that is provided for sequential programs, shared memory programs use tagged pointers to protect the memory associated with each process from being accessed by other processes.

4.3 V-Threads

V-Threads are used as the unit of scheduling processes. Each process requires a V-Thread slot somewhere in the machine to be able to execute. The run-time system will need to provide some mechanism to ensure that all processes are executed.

```
double U[XDIM][YDIM];                    /* U - dependent variables */
double norm;                             /* norm associated with block */

int main()
{   double termination;

    BlockInitialize();                   /* initialize a block */
    termination = BlockNorm()*EPSILON;   /* terminate when norm reduced */
    while( BlockNorm() > termination )   /* until termination */
        BlockTimestep();                 /* execute timesteps on block */
    return 0;                            /* then exit program */
}

void BlockInitialize()
{   int i,j;

    for( i = 0; i < XDIM; i++ )          /* initialize interior */
        for( j = 0; j < YDIM; j++ )
            U[i][j] = 0.0;
    for( j = 0; j < YDIM; j++ ) {        /* initialize j boundaries */
        U[0][j] = BOUNDARY;
        U[XDIM-1][j] = BOUNDARY;
    }
    for( i = 0; i < XDIM; i++ ) {        /* initialize i boundaries */
        U[i][0] = BOUNDARY;
        U[i][YDIM-1] = BOUNDARY;
    }
    norm = 2*BOUNDARY*(XDIM+YDIM-1);     /* sum of boundary values */
}

void BlockTimestep()
{   int i, j;
    double u1, u2, anorm=0.0;

    for( j = 1; j < YDIM-1; j++ )        /* sweep vertically */
        for( i = 1; i < XDIM-1; i++ ) {  /* sweep horizontally */
            u1 = U[i][j];                /* save old U(i,j) */
            u2 = (U[i+1][j] + U[i-1][j] +  /* add j neighbors to */
                  U[i][j+1] + U[i][j-1])/4.0;  /* i neighbors & average */
            U[i][j] = u2;                /* store new U(i,j) */
            anorm += fabs(u2-u1);        /* accumulate new norm */
        }
    norm = anorm;                        /* associate norm with block */
}

double BlockNorm()
{   return norm;
}
```

Program 1: Sequential Dirichlet Code

4.4 H-Threads

The compiler will do the best job it can of using all four H-Threads within the V-Thread that is executing the program. A significant amount of this usage will be attained through using loop unrolling

4.5 Example

The code shown in Program 2 is a shared-memory version of Program 1. Only the parts of the code which have changed are shown. Note that the global variable are marked as **shared** to indicate that there is only one copy of these arrays for the entire computation and they are shared between processes. Global variables which are not marked **shared** are one-copy-per-process. When the program is loaded, a specified number of processes are created. The number of processes remains fixed throughout the lifetime of the job. It is possible to have more than one process for each processor. When the program begins execution, all processes begin executing at **main()**.

The functions **Left()** and **Right()** need to be written by the programmer to ensure that each process accesses a disjoint portion of the shared arrays, and that the entire array is accessed. These functions return to the caller the bounds of an array which are treated as being local to a given process. To facilitate construction of these processes, and to allow selective serialization of code, the functions **Process()** and **Processes()** are added. These functions return, respectively, the process ID of the calling process, and the number of processes in the job. Synchronization is performed using the function **Barrier()**.

```
shared double U[XDIM][YDIM];                    /* U - dependent variables */
shared double norm[processes()];                /* norm associated with block */

void BlockInitialize()
{   int i,j;

    for( i = Left(); i <= Right(); i++ )        /* initialize interior */
        for( j = 0; j < YDIM; j++ )
            U[i][j] = 0.0;
    for( j = 0; j < YDIM; j++ ) {               /* initialize j boundaries */
        if( !Leftcut() ) U[Left()][j] = BOUNDARY;
        if( !Rightcut() ) U[Right()][j] = BOUNDARY;
    }
    for( i = Left(); i <= Right(); i++ ) {      /* initialize i boundaries */
        U[i][0] = BOUNDARY;
        U[i][YDIM-1] = BOUNDARY;
    }
    norm[Process()] = 2*BOUNDARY*(XDIM+YDIM-1);
    Barrier();                                  /* wait until all processes are done */
}

void BlockTimestep()
{   int i, j;
    int istart = Left() + ( Leftcut() ? 0 : 1 ),
    iend =Right() - ( Rightcut() ? 0 : 1 );
    double u1, u2, anorm = 0.0;
    double Uleft[YDIM], Uright[YDIM], Tleft, Tright;

    if( Leftcut() )                             /* get left boundary values */
        for(j = 0; j < YDIM; j++)
            Uleft[j] = U[Left()-1][j];
    if( Rightcut() )                            /* get right boundary values */
        for(j = 0; j < YDIM; j++)
            Uright[j] = U[Right()+1][j];
    Barrier();                                  /* wait until all boundaries have been copied */
    for( j = 1; j < YDIM-1; j++ )               /* sweep vertically */
    for( i = istart; i <= iend; i++ ) {         /* sweep horizontally */
        Tright = ( i < Right() ? U[i+1][j] : Uright[j] );
        Tleft = ( i > Left() ? U[i-1][j] : Uleft[j] );
        u1 = U[i][j];                           /* save old U(i,j) */
        u2 = (Tright + Tleft + U[i][j+1] + U[i][j-1])/4.0;  /* sum & average i,j neighbors */
        U[i][j]=u2;                             /* store new U(i,j) */
        anorm += fabs(u2-u1);                   /* accumulate new norm */
    }
    norm[Process()] = anorm;                    /* store copy of local norm */
    Barrier();                                  /* wait until sub-blocks have been updated */
    if( Process() == 0 )                        /* sum local norms */
        for( j = 1; j < Processes(); j++ )
            norm[0] += norm[j];
    Barrier();                                  /* wait until new global norm has been computed */
    norm[Process()] = norm[0];                  /* get new global norm */
}

double BlockNorm()
{   return norm[Process()];
}
```

Program 2: Shared Memory Dirichlet Code

5 Support for Message-Passing

Message passing provides explicit movement of data and mapping of control.

In the previous sections, access to non-local data is transparently provided by the compiler and run-time system. Unfortunately, the fine granularity of data transport may limit program performance for many applications. Explicit data movement or pre-fetch primitives increase this granularity to meet the needs of the application, so that acceptable performance can be obtained.

The message-passing system provided will be backwards-compatible with MDC on the J-Machine for mapping of control. Movement of data will be supported by standard MPI message-passing primitives [9]. In addition, we plan on providing library routines for efficient global operations. Finally, a graph library will be used to support irregular and/or adaptive computations by creating a network of arbitrarily connected processes.

The basic message-passing model is based on the concept of a process. Control is mapped to specific processes by creating threads within the process using the mapping annotation @ as in MDC; data is mapped to processes using MPI. The correspondence between processes and hardware processors may be static and/or dynamic. The initial number of processes present in the machine can be specified by the user at either link or load time.

5.1 Global Virtual Address Space

The global virtual address space will be used to support the distribution of code as shown in Figure 4. All global variables will be treated as one-copy-per-process with no coherence. Transmission of data between processes will be handled via explicit message-passing. By default, all code in a program will be treated as distributed, and attempts to fetch code which is not physically local will result in implicit communication. The notion of function replication described in [7] will be supported by mapping the range of global memory containing specific functions into physically local memory at all nodes in a partition. This same technique of virtual-to-physical mapping will be used to support global variables.

A process consists of a contiguous piece of virtual memory containing the address space of all of the code within a program. Each process also has a pointer to the storage for global variables within the process. Access to global variables will always be indirectly through this pointer. If possible, the address ranges for the global variables and the replicated functions are mapped into physically local memory. The physical storage for the distributed functions is divided amongst the nodes in a partition.

When a process is created, it contains a pointer to the global variables, and pointers to storage for a context and a stack. Both the context and stack are ideally completely resident in physically local memory. However, if there is not sufficient local memory

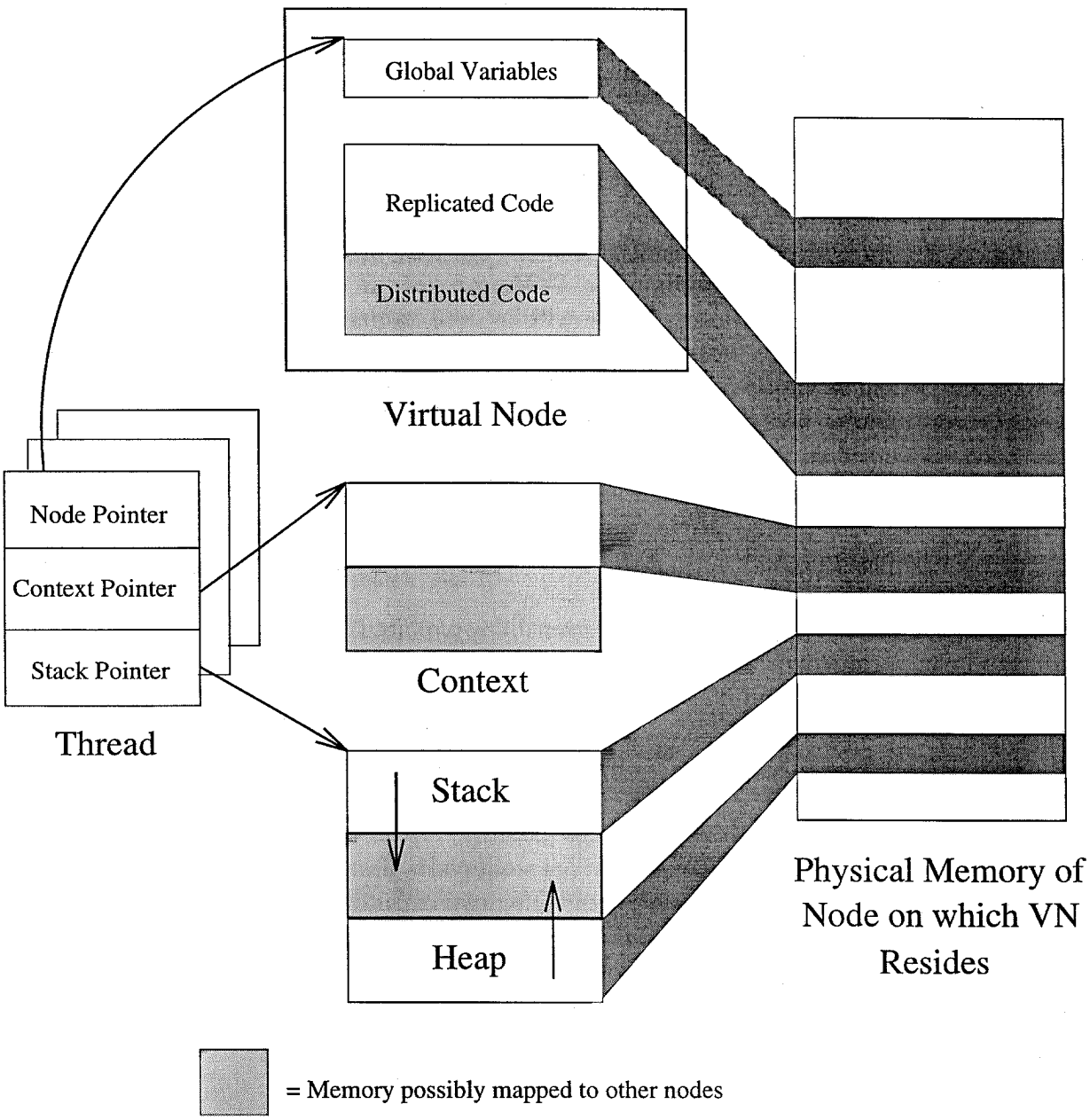

Figure 4: Mapping an MDC Program

available, either one of these data items may have some of its storage mapped to physical memory at another node.

5.2 Tagged Pointers

Tagged pointers will be used to provide memory protection between jobs and processes. They will also be used to protect code from being overwritten.

5.3 V-Threads

Each process is subdivided into threads. If there are four or fewer active threads at a given processor, they will each occupy one *V-Thread slot.* If there are more than four active threads, a software scheduling queue will be used to manage those threads which are not resident in a *V-thread slot.* Once a thread is loaded into a *slot*, it will be allowed to run to completion. A timeout interrupt will allow the runtime system to evict a thread that has ceased making progress. As long as one of the *H-threads* within the process is able to issue instructions, the thread will not be evicted.

5.4 H-Threads

As with the other models supported, *H-threads* will be exploited by the compiler to provide instruction-level parallelism.

5.5 Example

The code shown in Program 3 is a message-passing version of Programs 1 and 2. In this code, the global variables are one copy per node variables. The functions which are added to this version are **Send()** and **Recv()** for moving data between processes, and **Globalsum()** which gets a value from all processes, sums them, and returns the result to all processes. As with Program 2, this program is written to be executed in all processes. Message-passing programs are able to either have **main()** started at all processes, or to use the MDC model which starts **main()** only at process 0.

6 Hybrid Programs

An important aspect of this methodology is that it is not necessary to choose between programming models on a program-by-program basis. Programs can be constructed so that the most appropriate paradigm is used for each phase of the computation.

```
double U[Right() - Left() + 1][YDIM];                    /* U - dependent variables */
double norm;                                             /* norm associated with block */

void BlockInitialize()
{   int i,j;

    for( i = Left(); i <= Right(); i++ )                 /* initialize interior */
        for( j = 0; j < YDIM; j++ )
            U[i][j] = 0.0;
    for( j = 0; j < YDIM; j++ ) {                        /* initialize J boundaries */
        if( !Leftcut() ) U[Left()][j] = BOUNDARY;
        if( !Rightcut() )U[Right()][j] = BOUNDARY;
    }
    for( i = Left(); i <= Right(); i++ ) {               /* initialize I boundaries */
        U[i][0] = BOUNDARY;
        U[i][YDIM-1] = BOUNDARY;
    }
    norm = 2*BOUNDARY*(XDIM+YDIM-1);                     /* sum of boundary values */
}

void BlockTimestep()
{   int i, j;
    int istart = Left() + ( Leftcut() ? 0 : 1 ),
        iend = Right() - ( Rightcut() ? 0 : 1 );
    double u1, u2, anorm = 0.0;
    double Uleft[YDIM], Uright[YDIM], Tleft, Tright;

    if( Leftcut() )                                      /* send left boundary values */
        Send(U[Left()], YDIM, (Process() - 1) % Processes());
    if( Rightcut() )                                     /* send right boundary values */
        Send(U[Right()], YDIM, (Process() + 1) % Processes());
    if( Leftcut() )                                      /* receive left boundary values */
        Recv(Uleft, YDIM, (Process() - 1) % Processes());
    if( Rightcut() )                                     /* receive right boundary values */
        Recv(Uright, YDIM, (Process() + 1) % Processes());
    for( j = 1; j < YDIM-1; j++ )                        /* sweep vertically */
        for( i = istart; i <= iend; i++ ) {              /* sweep horizontally */
            Tright = ( i < Right() ? U[i+1][j] : Uright[j] );   /* get right U value */
            Tleft = ( i > Left() ? U[i-1][j] : Uleft[j] );      /* get left U value */
            u1 = U[i][j];                                       /* save old U(i,j) */
            u2 = (Tright + Tleft + U[i][j+1] + U[i][j-1])/4.0;  /* sum & average i,j neighbors */
            U[i][j]=u2;                                         /* store new U(i,j) */
            anorm += fabs(u2-u1);                               /* accumulate new norm */
        }
    Globalsum(&anorm, &norm, 1);                         /* calculate new global norm */
}

double BlockNorm()
{   return norm;
}
```

Program 3: Message-Passing Dirichlet Code

As an example of the type of intermediate program that results from this methodology, Program 4 is a hybrid shared memory/message-passing version of the example program. In this example, each process operates on a particular portion of the shared **U** array, based on the index extents provided by the **Left()** and **Right()** functions. In addition, the **Leftcut()** and **Rightcut()** functions allow a process to determine if its portion of **U** is on a boundary, so that it may adjust its initialization and boundary exchange behavior accordingly. In **BlockTimestep()**, the nodes exchange boundary elements via shared memory accesses. A **Barrier()** call is performed to guarantee that all boundaries have been exchanged before the computation is continued. Finally, a global norm is calculated via **Globalsum()**. Both **Barrier()** and **Globalsum()** may be implemented using message-passing for efficiency.

7 Coherency

To support the *global virtual address space*, one must provide some mechanism for ensuring coherency of shared data objects. This can be done by maintaining a bit-field of accessors for each cache line within a shared page [4]. When the page is modified, only those processors that had actually obtained copies of the cache line would be notified of the change. While this scheme is scalable in terms of limiting communication overhead, it does not appear scalable in terms of storage overhead. On a thousand node machine, the access lists would require 1024 words for a 512-word page. On a million node machine this does not appear desirable.

It has been our experience that once data/control decompositions have been performed, the maximum number of nodes requiring access to a shared data object is application-specific and *largely independent* of the machine size. For example, in a two-dimensional domain decomposed into blocks, only the processors governing subdomains adjoining a particular block would require access to that block. Consequentially, it makes sense to provide exact update lists for a only a fixed number of nodes. For example, one might maintain a list of up to 32 accessors for each cache line. If the number of accessors exceeds 32, then one would use a *fixed-size* bit field to encode *groups* of accessors. For example, on a 512-node machine, each bit of a 64-bit group accessor field would represent an 8-node group. If any node within that group were to access a cache line, then all the nodes within the group would be notified if the cache line were to change.

In this case, assuming the accessor ID's are stored as 16-bit integers, the storage overhead would be 512 words, but would not grow with the machine size. In short, *we believe it is more appropriate to optimize for programs which have been decomposed rather than for those which have not.*

```
shared double U[XDIM][YDIM];                    /* shared grid */
double norm;                                    /* local norm */

int main()
{   double termination;

    BlockInitialize();                          /* initialize a block */
    termination = BlockNorm()*EPSILON;          /* terminate when norm reduced */
    while( BlockNorm() > termination )          /* until termination */
        BlockTimestep();                        /* execute timesteps on block */
    return 0;                                   /* then exit program */
}

void BlockInitialize()
{   int i,j;

    for( i = Left(); i <= Right(); i++ )        /* initialize interior */
        for( j = 0; j < YDIM; j++ )
            U[i][j] = 0.0;
    for( j = 0; j < YDIM; j++ ) {               /* initialize j boundaries */
        if( !Leftcut() ) U[Left()][j] = BOUNDARY;
        if( !Rightcut() ) U[Right()][j] = BOUNDARY;
    }
    for( i = Left(); i <= Right(); i++ ) {      /* initialize i boundaries */
        U[i][0] = BOUNDARY;
        U[i][YDIM-1] = BOUNDARY;
    }
    norm = 2*BOUNDARY*(XDIM+YDIM-1);
    Barrier();                                  /* wait until all processes are done */
}

void BlockTimestep()
{   int i, j;
    int istart = Left() + ( Leftcut() ? 0 : 1 ),
    iend =Right() - ( Rightcut() ? 0 : 1 );
    double u1, u2, anorm = 0.0;
    double Uleft[YDIM], Uright[YDIM], Tleft, Tright;

    if( Leftcut() )                             /* get left boundary values */
        for(j = 0; j < YDIM; j++)
            Uleft[j] = U[Left()-1][j];
    if( Rightcut() )                            /* get right boundary values */
        for(j = 0; j < YDIM; j++)
            Uright[j] = U[Right()+1][j];
    Barrier();                                  /* wait until all boundaries have been copied */
    for( j = 1; j < YDIM-1; j++ )               /* sweep vertically */
    for( i = istart; i <= iend; i++ ) {         /* sweep horizontally */
        Tright = ( i < Right() ? U[i+1][j] : Uright[j] );
        Tleft = ( i > Left() ? U[i-1][j] : Uleft[j] );
        u1 = U[i][j];                           /* save old U(i,j) */
        u2 = (Tright + Tleft + U[i][j+1] + U[i][j-1])/4.0;  /* sum & average i,j neighbors */
        U[i][j]=u2;                             /* store new U(i,j) */
        anorm += fabs(u2-u1);                   /* accumulate new norm */
    }
    norm = Globalsum(anorm);                    /* calculate new global norm */
}

double BlockNorm()
{   return norm;
}
```

Program 4: Hybrid Shared Memory/Message-Passing Dirichlet Code

References

[1] Amarasinghe, S.P., Anderson, J.M., Lam, M.S. and Tseng C.W., "The SUIF Compiler for Scalable Parallel Machines", Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, February, 1995.

[2] Cooper, K.D., Hall, M. W., and Kennedy, K., "A Methodology for Procedure Cloning" Computer Languages, 19(2), April 1993, pages105-118.

[3] Dally, W. J., et al., "The J-Machine: A Fine-grain Concurrent Computer," *Information Processing 89*, G. X. Ritter (ed.), Elsevier Science Publishers B.V., North Holland, IFIP, 1989.

[4] Dally, W. J., et al., "M-Machine Architecture v1.0," Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Concurrent VLSI Architecture Memo 58, February, 1994.

[5] Fox, Geoffrey, et. al., *Fortran D Language Specification*, Technical Report, Center for Research on Parallel Computation, CRPC-TR-90079, 1990.

[6] High Performance Fortran Forum, *High Performance Fortran Language Specification, Scientific Programming*, June, 1993.

[7] Maskit, Daniel, and Taylor, Stephen, "A Message-Driven Programming System for Fine-Grain Multicomputers", *Software - Practice and Experience*, 24, 953-980 (1994).

[8] Maskit, D., et. al., *System Tools for the J-Machine*, California Institute of Technology, Department of Computer Science Technical Report, CS-TR-93-12, 1993.

[9] Message-Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, May, 1994.